

**VM-Regress: A VM Regression and Benchmarking Tool For  
Linux  
Version 0.9**

---

Mel Gorman  
Email: [mel@csn.ul.ie](mailto:mel@csn.ul.ie)

*Computer Science and Information Systems, University of Limerick*  
Generated: April 18, 2003

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Kernel Modules</b>	<b>6</b>
2.1	Core Module Services . . . . .	7
2.2	Proc Buffer Management . . . . .	15
2.3	Module Structure . . . . .	18
2.4	Sense Modules . . . . .	21
2.5	Test Modules . . . . .	24
2.6	Benchmark Modules . . . . .	29
<b>3</b>	<b>Userland Scripts</b>	<b>31</b>
3.1	Physical Page Test . . . . .	32
3.2	Page Fault Test . . . . .	32
3.3	Anonymous and File Based Faulting Benchmark . . . . .	33
<b>4</b>	<b>Perl Support Libraries</b>	<b>37</b>
4.1	VMR::External . . . . .	37
4.2	VMR::File . . . . .	38
4.3	VMR::Graph . . . . .	38
4.4	VMR::Kernel . . . . .	39
4.5	VMR::Pagemap . . . . .	39
4.6	VMR::Reference . . . . .	40
4.7	VMR::Time . . . . .	41
4.8	VMR::Report . . . . .	41
<b>5</b>	<b>Interpreting Results</b>	<b>43</b>

# Abstract

The Linux Virtual Memory (VM) subsystem has changed considerably during recent years. The changes have been in response to real world program behaviour, empirical evidence and developer intuition. Unfortunately, what has not developed is a reliable way to test and benchmark the VM. User reported behaviour is often difficult to reproduce and depends heavily on the developer guessing what may have happened. Tests for changes are minimal at best and non-existent at worst. The most common test appears to be stress testing which cannot be reliably expected to test all code paths. Changes that are marked "Untested" are common.

VM-Regress is a tool that is aimed at providing a reliable way to test and benchmark the VM and currently is a basic framework which more comprehensive tests may be built upon. It uses a combination of userspace scripts and kernel modules to determine exactly how a test will run and what it will be testing. This allows reliable, and more importantly reproducible tests that developers and users can work with.

# Chapter 1

## Introduction

VM-Regress is a tool for testing and benchmarking specific parts of the Linux Virtual Memory (VM) system. Unlike other test suites and benchmarks, it is extremely specific in what it is testing and uses kernel modules as well as userland programs to ensure repeatable behaviour and to collect data. The aim is to use userspace programs for collecting data, generating reports and to “drive” the test. The modules provide fine grained control of the system as well as the potential to extract information on the current running kernel which is unavailable from userspace.

Repeatable and microscopic tests are both an advantage and a disadvantage. The principle advantage is that its behaviour is designed and known in advance of the test. This allows a group of developers to run the same test and be sure their systems all behaved approximately the same and can be sure of the affected code paths. Current tests which impact a wider range of code and isolating bottlenecks is a considerable challenge.

The second main advantage is that a developer can make sure a section of the kernel still works after making changes. For example, if a developer altered the buddy allocator or replaced it with a new physical page allocator, the `alloc` module could be used to test as many of the code paths as possible. If the test completes, the developer can be reasonably sure it is working reliably. Without the tool, the developer would have to use the system as normal and wait for issues to develop.

The tool does have disadvantages that should be noted. The first is that a tool that examines such small parts of the kernel very little information about overall system performance. Individual tests can be run on page swapping, page allocation, memory mapping and so on. These tests combined give information about each system, but it would take a very skilled developer to know how well a database server would behave based on the figures. The tests only show individual subsystem behaviour, they don’t show how well they interact. For that, userland benchmarking tools or real programs are required.

The benchmark tools are aiming towards measuring overall performance as well as the performance of individual sections. At time of writing, the benchmark modules provide interfaces for a userland program to collect information from the kernel. It is hoped they will develop to the point where certain library functions can be overloaded with `LD_PRELOAD` tricks to fool normal applications into using VM Regress into collecting data.

The second disadvantage is collecting some of the data is tricky. A kernel module can-

not even easily print to console where a userspace program could use `printf` or equivalent function. More importantly, a programming error in a kernel module leads to crashed systems, not segmentation faults. This means that the modules need to keep a pool of memory available for storing test results. Alternatively the userspace program has to keep collecting the data and storing it to disk.

The whole test is not implemented as kernel modules as the development cost of using kernel modules far too high and there is a number of operations which are simply impractical to perform in kernel space. The kernel modules are broken up into four major headings, **core**, **sense**, **test** and **bench**. Sense modules tell what is going on in the kernel at the moment. Test modules execute a specific code path or linear test and bench modules emulate real world behaviour. Core modules provide basic functions common to many modules. This includes functions such as managing buffers for printing test results and walking page tables.

The userland collection programs are Perl scripts. They try to have as minimal an impact on the system as possible so as not to skew the results. For each of the test and benchmark kernel modules, there is a script which can run the test and generate a report to simplify the data collection with both kernel and userspace.

In summary, the tool is the beginning of the first tool of it's kind. A full VM regression and benchmarking tool. In time, it is hoped that it will be comprehensive enough to be able to provide empirical performance figures for all aspects of the VM as well as proving beyond reasonably doubt the VM is stable for any given kernel release.

# Chapter 2

## Kernel Modules

This chapter describes the kernel modules which give fine grained control and access to the kernel structures. This chapter is mainly of interest only to developers of new tests as userspace scripts already exist to use most module functionality.

The first principle of the kernel modules is to be as simple as possible and provide only a small specific task. Each module individually is not particularly useful, it is the responsibility of userspace to use the modules in a suitable combination.

The second principle is to have the highest level module as simple as possible. Most of the complex work of kernel modules is handled by the core modules and header files as they are common to all modules. All modules use the same initialisation code (`init.c`), `/proc/` printing code (`proc.c`), pagetable walk functions etc (all in the `core/` directory) making the final test modules very small. For example, the module for printing out all the zone information (`zone.c`) is only 125 lines long which is mainly comments and includes. The ideal is that the developer of a new test can just use the framework without worrying about how it works.

## 2.1 Core Module Services

This section will describe how the basic framework of the kernel structures are managed. It will begin by describing what modules depend on each other and some of the services that are provided by the core. We will then describe how the core is used to manage proc buffers.

### 2.1.1 Module Dependencies

The module dependency for the tool is very straight forward as can be seen from Figure 2.1. It is expected that the dependency for the tool as it develops will continue to be flat like this.

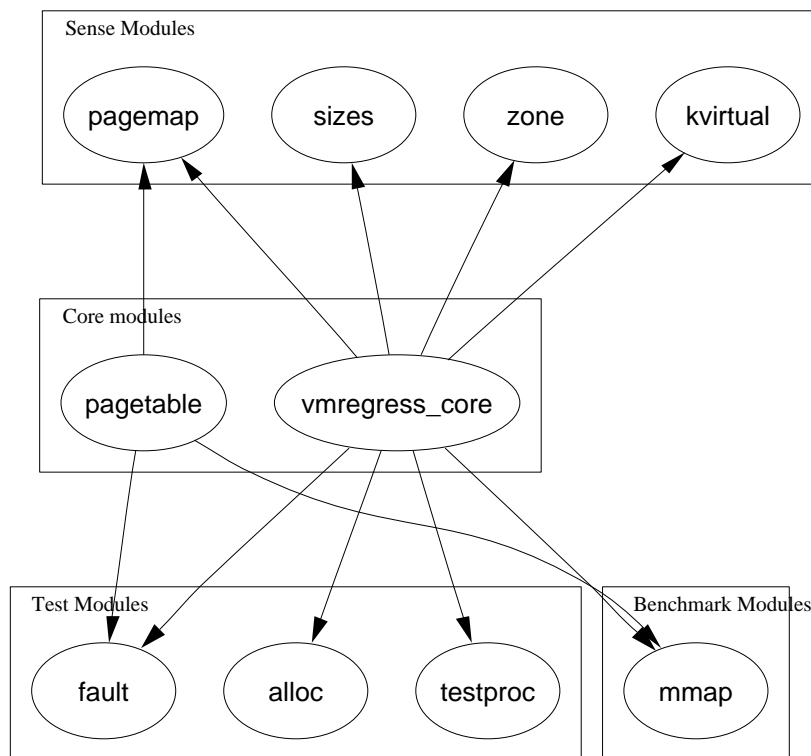


Figure 2.1: Module Dependency Graph

What is omitted from the graph is the initialisation and proc printing code. All modules include `src/init/init.c` which has the initialisation code for each module. Each module needs a slightly different initialisation and to it is `#included` with `#defines` indicating what type of code is required. It is similar for the proc reading functions which include `src/init/proc.c`. This sounds a little unconventional but it drastically reduces the amount of duplicated code in the system due to minor variations.

## 2.1.2 *vmregress\_core* Structure

*vmregress\_core* provides basic infrastructure that all modules need. The functionality includes

- Defines a structure for defining tests
- Creating the `/proc/vmregress` directory
- Allocating, freeing and growing buffers for proc entries
- Macros for printing to buffers
- Acquiring a pointer to `pgdat_list`
- Basic string functions
- Basic timing macros
- Calling `schedule()` when necessary

We will discuss each of these functions in turn in the coming sections.

## 2.1.3 Declaring Tests

Each module is expected to declare what its name is with a `#define` called `MODULENAME`. For every test or proc interface the module exposes, it is expected to have a `struct vmr_desc` describing information about the test.

All the information required for the test is included in this struct and it mainly concerns itself with the maintenance of proc buffers which are needed to print out test results. Concurrent processes can use the same test modules but only *one* is allowed to write at a time as modules have no easy means of outputting to several streams like userspace can<sup>1</sup>.

The struct is defined in `<vmregress_core.h>` as

---

<sup>1</sup>Yes, it could be implemented to allow files to be opened from the module but I think it would be pushing too much work into the kernel modules for something that can be handled from userspace



```

30 typedef struct vmr_desc {
31     /* Lock to data */
32     spinlock_t lock;          /* Lock to protect struct. Mainly
33                                * of importance to setting the
34                                * pid of who is writing. Many
35                                * tests can run but only one
36                                * PID may write to the buffer
37                                */
38
39     /* Proc entry info */
40     int procentry;            /* Index of proc buffer been written to */
41     char *procbuf;           /* Buffer to print to */
42     int procbuf_size;        /* Buffer size */
43     int written;             /* Bytes written to buffer */
44     pid_t pid;               /* PID of the test writer */
45
46     /* Persistent info */
47     unsigned long mapaddr;    /*
48                                * Address of a memory mapped
49                                * area. This is needed for
50                                * printing out pages present
51                                * or swapped within a region.
52                                * See pagetable.c:vmr_printpage
53                                */
54
55     /* Test configuration */
56     char name[40];           /* Name of the test */
57     unsigned long flags;     /* Bitmap of test flags */
58
59     /* Read/Write procedures for this proc entry */
60     int (*read_proc)(char *buf, char **start,
61                      off_t offset, int count, int *eof, void *data);
61     int (*write_proc)(struct file *file, const char *buf,
62                       unsigned long count, void *data);
63
64 } vmr_desc_t;

```

A brief description of the fields is as follows:

**lock** When a module needs to open a proc buffer, it calls `vmrproc_openbuffer()` which uses this spinlock to protect the struct while it is checked for concurrent accesses;

**procentry** A module may have multiple proc entries which means multiple test descriptions in the `testinfo` array (Described later in this section). The index of this

test is recorded in **procentry** and passed in as private data to the proc read/write functions;

**procbuf** Each test has an individual proc buffer to write to with is allocated with **vmrproc\_allocbuffer()**;

**procbuf.size** This is the size of the proc buffer in bytes. If the proc buffer would be overwritten, it will be grown unless the **VMR\_NOGROW** flag is specified;

**written** This is how many bytes have been written to the proc buffer so far;

**mapaddr** Eventually tests will take place on memory areas or require a dedicated memory are for the test. Currently there is only one, **fault.o** which needs this field when printing out the page present/swapped information;

**name** This is a human readable name for the test modules proc entry. The name for the test that appears in **/proc/vmregress** is generally something like **MODULENAME\_name**;

**flags** A test can have a number of flags which determine behavior which can be stored here.

As the core depends on a number of macros to hide proc printing and some core functions, the name of the test descriptor array is important. While generally speaking this is bad coding practice, the emphasis was on making the main modules as readable as possible and the reserved names are very rare.

The test descriptor array must be called **testinfo** and is obviously of type **vmr\_desc\_t**. By adhering to this name, the printing macros described later take very few parameters and makes the code look simpler. A macro **VMR\_DESC\_INIT()** is provided to statically initialise the struct. The module for testing page faults for instance describes it's tests as follows

```
86 #define MODULENAME "test_alloc"
87 #define NUM_PROC_ENTRIES 4
88
89 /* Tests */
90 #define TEST_FAST 0
91 #define TEST_LOW 1
92 #define TEST_MIN 2
93 #define TEST_ZERO 3
94
95 static vmr_desc_t testinfo[] = {
96     VMR_DESC_INIT(TEST_FAST, MODULENAME "_fast", vmr_read_proc, vmr_write_proc),
97     VMR_DESC_INIT(TEST_LOW, MODULENAME "_low", vmr_read_proc, vmr_write_proc),
98     VMR_DESC_INIT(TEST_MIN, MODULENAME "_min", vmr_read_proc, vmr_write_proc),
99     VMR_DESC_INIT(TEST_ZERO, MODULENAME "_zero", vmr_read_proc, vmr_write_proc)
100 };
```

MODULENAME has already been discussed. NUM\_PROC\_ENTRIES is needed for the module initialisation code so it knows how many tests to look for. The TEST\_\* are the indexes into the testinfo array<sup>2</sup>.

The array testinfo is specific to this module (hence static) and has four proc entries. It takes four parameters. The first is the index into the array. The second is the name that will appear in the proc directory; for example, the first proc entry will be test\_alloc\_fast. The third parameter is function to call if the proc entry is read (vmr\_read\_proc() is in proc.c) and the fourth parameter is the write procedure.

The read and write procedures have a small side effect. If a read procedure is not defined, it is presumed the proc entry is actually a directory. This is a special case used only by vmregress\_core and is not for general use. The write procedure is strictly optional but if one module uses it, NUMBER\_PROC\_WRITE\_PARAMETERS must be defined so that the proc procedures know how many input variables to take.

## 2.1.4 The /proc/vmregress Directory

The core is responsible for creating this proc entry. Once it is created, it exports the symbol vmregress\_proc\_dir so other modules have a handle to it. The handle to the directory is stored in vmregress\_core and exported to other modules as a symbol.

## Test Flags

Each test has a number of flags which determine test behavior. The flags are all defined in <vmregress\_core.h> and are described in Table 2.1

Flag	Description
VMR_PRINTMANY	Normally multiple processes are not allowed to write to the one buffer. If this is set, multiple processes can and the PID of the writing process will be pre-pended to the proc output
VMR_WAITPROC	If a process tries to open a buffer already in use, printing is just disabled for that process. If this flag is set, it will instead block waiting for the buffer to be free
VMR_NOGROW	Proc buffers normally grow dynamically if required. If this flag is set, the proc buffer will simply be disabled when it is full
VMR_PRINTMAP	If set, a map representing present pages in a mapped area will be printed out. This is largely for future needs when the mapaddr field is used more

Table 2.1: VM Regress Test Flags

---

<sup>2</sup>Yes, they really should be an enumerated type with enum but they aren't and the change would not be world shaking

## 2.1.5 Acquiring `pgdat_list`

For many tests, a handle to the top level memory node struct is required. In the core kernel, this is simple as the variable `pgdat_list` gives a pointer to the head of a linked list of all memory nodes but this symbol is not exported to modules. The function `get_pgdat_list()` is provided to get a handle to `pgdat_list`.

```
/**
 *
 * get_pgdat_list - Return the pgdat_list
 * @page - A page provided to track the parent pgdat_list if necessary
 */
pg_data_t *get_pgdat_list(void);
```

A kernel patch is available with VM-Regress that exports `pgdat_list` correctly. If this is not applied a workaround is used. The workaround uses a page to find the `zone_t` it belongs to. The `zone_t` struct points to it's parent `pg_data_t`. The workaround does not guarantee that all memory nodes will be visible but on many architectures like the x86, there is only one node available so it is not a problem. A warning will be printed to syslog but it can be safely ignored on machines with only one memory node.

## 2.1.6 Basic String Handling Functions

Only one string function is exported by the core. It is an implementation for `strtol` for converting strings to integers called `vmr_strtol()`. It is needed for reading information via `/proc`.

## 2.1.7 Basic Timing Macros

Some tests try and print out how long the test took. A macro `jiffies_to_ms()` gives a very rough time based on jiffies variable. The macro is defined in `<vmregress_core.h>` as:

```
250 #define jiffies_to_ms(start) ((1000 * (jiffies - start)) / HZ)
```

The parameter it expects is the jiffy count the test started at. Note that this method of timing is fairly inaccurate and the granularity is pretty poor. It could be greatly improved if the TSC was used.

## 2.1.8 Calling `schedule()`

A test can last any length of time. If `schedule()` was not called, it is possible the whole system could be locked up. Each test should the macro `check_resched()` periodically which in turn calls the function `check_resched_nocount()`.

```
int check_resched_nocount(void);
#define check_resched(counter) if (check_resched_nocount() == 1) counter++
```

The function returns 1 if `schedule()` was called and 0 otherwise. This allows tests to keep a running count of how many times it was necessary for them to schedule themselves.

## 2.1.9 Page Table Walking

The `pagetable` module is concerned with page table traversal. In the core kernel, it is far more efficient to have a number of inline functions for this type of work but in VM-Regress, it would lead to too much duplicated code. Here, code clarity and simplicity is more important than speed. There is four functions exported.

The first function is `get_struct_page()` which returns the `struct page` representing `addr` in the current `mm_struct`. It is not used in any module but is useful during development of modules.

```
/**
 * get_struct_page - Gets a struct page for a particular address
 * @address - the address of the page we need
 */
struct page *get_struct_page(unsigned long addr);
```

The second function is `forall_pte_mm()` which is much more useful. It will execute a callback function `func()` and pass in private data for every `pte()` within an address range. It will return the sum of all values returned by `func()`. This is useful for such things as counting present `ptes` or for touching all pages within an address range. While walking the page table, the `page_table_lock` is held but is released before using the callback function. It is defined as

```
/**
 * forall_pte_mm - Execute a function func for all pages within a range
 * @mm: The memory area been examined
 * @addr: The starting address
 * @len: The size of the area to count pages in
 * @sched_count: A running count of how many times schedule() was called
 * @data: Pointer to caller data
 * @func: The function to call
 *
 * This function presumes it will be called for an addr and len
 * with a valid vma.
 */
unsigned long forall_pte_mm(struct mm_struct *mm, unsigned long addr,
                          unsigned long len, unsigned long *sched_count,
                          void *data,
                          unsigned long (*func)(pte_t *, unsigned long, void *))
```

A sample declaration of a callback function is `ispage_present()` used to count how many `pte()`s within an address range are present in memory. The function is not exported.

```

/**
 * ispage_present - Returns 1 if a pte is present in memory
 * @pte: The pte been examined
 * @addr: The address the pte is at (unused)
 * @data: Pointer to user data (unused)
 *
 * This is a callback function for forall_pages_mm() to use.
 */
unsigned long ispage_present(pte_t *pte, unsigned long addr, void *data)

```

The third function provided by this module is `countpages_mm()`. It uses `forall_pages_mm()` to count how many pages are present within an address range

```

/**
 * countpages_mm - Count how many pages are present in a mm
 * @mm: The mm to count pages in
 * @addr: The starting address
 * @len: The length of the address space to check
 * @sched_count: A count of how many times schedule() was called
 */
unsigned long countpages_mm(struct mm_struct *mm, unsigned long addr,
                           unsigned long len, unsigned long *sched_count) {

    return forall_pte_mm(mm, addr, len, sched_count, NULL, ispage_present);
}

```

The last exported function is `vmr_printpage()`. This is of particular interest to modules which work with virtual memory areas. Given a virtual address and range, it will print an encoded map showing pages present or swapped out. Each character in the map represents 4 pages. The lower four bits (0-3) are set or clear depending on the presence of the page. Bits 4 and 5 are set to 1 so that the map will be semi-readable to humans.

```

/**
 * vmr_printpage - Sets the corresponding bit in the proc buffer (callback)
 * @pte: The pte been examined
 * @addr: The address the pte is at
 * @data: Pointer to user data (vmr_desc_t)
 *
 * This is the callback for the pagetable walk. It will set the appropriate
 * bit in the proc buffer. The beginning of the map is presumed to be
 * testinfo->written is pointing to
 */
unsigned long vmr_printpage(pte_t *pte, unsigned long addr, void *data)

```

## 2.2 Proc Buffer Management

Kernel modules cannot print directly to console. The closest equivalent is using `printk()` to print to syslog but that is not suitable for printing out test results. VM-Regress uses the proc interface to receive input and print out results. The proc filesystem provides a page sized buffer that a caller can write to. For the purposes of vmregress, this could be too small so larger buffers are created using `vmalloc()` and managed by the core. This section focuses on how proc buffers are allocated and freed, opened and closed and printed to.

### 2.2.1 Allocating/Freeing Proc Buffers

Three functions are concerned with the allocation, freeing and growing of proc buffers. They are all declared in `<procprint.h>`. The first is `vmrproc_allocbuffer()` which takes two parameters, the number of pages to allocate and the test descriptor.

If the allocation is successful, `testinfo→procbuf` will store the newly allocated buffer, its size will be in `testinfo→procbuf_size` and 0 will be returned. It is important to note that if a buffer already exists, it will be freed for you and old pointers will be invalid. If it fails to allocate a buffer `-ENOMEM` is returned and the old buffer, if it existed will still be valid.

```
/**
 * vmrproc_allocbuffer - Allocates a buffer for printing out proc information
 * @pages - number of pages to allocate
 * @testinfo - The test descriptor
 */
int vmrproc_allocbuffer(unsigned int pages, vmr_desc_t *testinfo)
```

The second is `vmrproc_freebuffer()` which just takes the test descriptor as a parameter. On return, `testinfo→procbuf` will be NULL and `testinfo→procbuf_size` will be 0. It cannot fail

```
/**
 * vmrproc_freebuffer - Frees the buffer used for printing proc information
 * @testinfo: The test descriptor
 *
 * This function will adjust the callers buffer and buffer size parameters. This
 * is handy if the size of the proc buffer is expected to change for the
 * lifetime of the module
 */
void vmrproc_freebuffer(vmr_desc_t *testinfo);
```

The last function, `vmrproc_growbuffer()` grows a proc buffer by the requested size. The grow function is provided in case a test needs a larger buffer. Most tests will not but if information about page presence is been printed, a quiet large buffer will be required so the buffer may be grown. It is very important to note that the buffer may change location

after the resizing and old pointers will be invalid. This will not affect the proc printing macros but it will invalidate any private pointers.

The parameters are the number of pages to grow by and the test descriptor. On success, 0 is returned. On failure -ENOMEM.

```
/**
 * vmrproc_growbuffer - Grow the proc buffer by a number of pages
 * @pages: The number of pages to grow by
 * @testinfo: The test descriptor struct
 *
 * This function will grow a buffer of a number of pages and copy in the
 * old contents. It is an expensive function so only use if you have to
 */
int vmrproc_growbuffer(unsigned int pages, vmr_desc_t *testinfo)
```

## 2.2.2 Opening/Closing Proc Buffers

As multiple processes may run the same test at the same time, it is possible a proc buffer could be contended by a number of processes. To prevent the inevitable collision, buffers must be opened with `vmrproc_openbuffer()`. On open, the buffer will be reserved exclusively for that process as the PID of the writing process is stored in the test descriptor.

If the buffer is already acquired when open is called, one of three things can happen. If no test flags are specified, open will return but any print by that process will be simply ignored and the test results lost. If `VMR_WAITPROC` is set in the test descriptor flags, then the opening process will block for up to five until the buffer is free. If `VMR_PRINTMANY` is set, multiple processes can open the buffer and each line is prepended with the writing process PID.

```
/**
 * vmrproc_openbuffer - Attempts to acquire a buffer and clears it
 * @testinfo: The test descriptor
 *
 * When a test begins, this function is called. The lock is acquired and
 * the PID examined. If the PID is 0, there is no writers so this process
 * gets it and is allowed to write. Callers should use vmrproc_closebuffer
 * to ensure the proc buffer is freed
 */
```

When the process is finished, it is expected to close the buffer with `vmrproc_closebuffer()` so that it is free for others to use. There is one case where modules are not expected to call `vmrproc_closebuffer()` themselves. If a module is using a read callback to have a function called every time the proc entry is read from userspace, it is expected that the callback will open the buffer and `vmr_read_proc()` will close the buffer when it has been fully read by userspace.



```
/**
 * vmrproc_closebuffer - Close access to a proc buffer
 * @testinfo: The test descriptor
 */
```

## 2.2.3 Printing to Proc Buffers

There is two macros provided related to printing. The first is `vmr_printk()` which is a wrapper around the `printk()` function to print out the priority and the module name. It is used to simply print out messages to syslog. It is defined as

```
100 #define vmr_printk(x,args...)      printk("<1>" MODULENAME ": " x, ## args)
```

The second macro is `printp()` and is used for printing to proc buffers. Unfortunately, as a macro, it has to make a number of presumptions. First, it presumes the test descriptor array of `vmr_desc_t` structs is called `testinfo` as described in Section 2.1.3.

Secondly, it presumes there is a variable called `procentry` which is the index in the `testinfo` struct to use. The macro is responsible for tracking how much has been written to the proc buffer. If the write will overflow the buffer, it will be dynamically grown and a message printed to syslog with `vmr_printk()`.

This seemingly clumsy sounding mechanism makes the code very simple to read and provides bounds checking at every print. It declares a number of local variables and calls the third macro `vmr_snprintf()` which does the actual printing and bounds checking.

## 2.3 Module Structure

This section will cover some characteristics of modules and how they are structured. All modules follow roughly the same format so once one module is well understood, the rest are much easier to follow. A template module is provided with the package called `template.c`. It performs most tasks a test module should do.

### 2.3.1 Module Preamble

This is generic module preamble. The top of each module has a comment describing what the module does. Next is `#includes` before the test is described as covered in Section 2.1.3.

### 2.3.2 Providing Help

Almost modules when loaded export an interface to `/proc`. Modules have the option of having a help message displayed in the proc buffer on load. This is convenient for having a quick reference to **cat** after the module is loaded to see what parameters it takes.

To have the help message displayed, a callback function which takes an index into the `testinfo` struct must be defined. This callback function is called by the initialisation code in `init.c` when the proc buffers are being allocated. The function should simply call `vmrproc_openbuffer()`, print out the help message with `printp()` and close the buffer again with `vmrproc_closebuffer()`.

To have `init.c` use the callback, a define called `VMR_HELP_PROVIDED` must exist with the name of the callback function. For example, the `alloc` module declares its callback as follows

```
448 #define VMR_HELP_PROVIDED test_alloc_help
```

### 2.3.3 Calculating Parameters

Different tests will have different parameters depending on the input argument. It is usually called `MODULENAME_calculate_parameters()`.

### 2.3.4 Reading Information from a Module

Some modules simply provide information when the proc entry is read such as the `zone` module which prints information on each zone in the system. As the proc buffer should change on each read, a callback function is declared with the define `VMR_READ_CALLBACK` which takes a procentry as a parameters. the `zone` module declares its callback as follows

```
123 #define VMR_READ_PROC_CALLBACK zone_getproc
```

### 2.3.5 Running a Test

A single function is responsible for running the test which is called by the proc write procedure. It is usually called `MODULENAME_runttest()` but `proc.c` needs a way of identifying it. It does this by having a define called `VMR.WRITE.CALLBACK` which is a function which takes a procentry as a parameter which is called when proc is written to. The `alloc` module declares its write callback as follows

```
445 #define VMR_WRITE_CALLBACK test_alloc_runttest
```

As each module could take a varying number of parameters on write, the module is responsible for declaring how many parameters it expects with `NUMBER_PROC_WRITE_PARAMETERS`. The `alloc` module expects two parameters so it is declared as

```
443 #define NUMBER_PROC_WRITE_PARAMETERS 2
```

Optionally a callback function may be declared with `CHECK_PROC_PARAMETERS` which will sanity check the parameters. It takes an array of integers and the number of parameters as an argument. the `alloc` module sanity checks its parameters by declaring

```
444 #define CHECK_PROC_PARAMETERS vmr_sanity
```

### 2.3.6 Module Initialisation

All kernel modules must declare an initialisation and cleanup routine with `module_init()` and `module_exit()`. Previously in VM Regress, all modules had their own routines but they were all essentially the same with minor differences. To cut down on the amount of duplicated code, it was all moved to `src/init/init.c` and is `#included` by each module. As each module requires slightly different code, a number of defines determine what type of code is used. The defines are listed as follows:

**VMR\_MODULE\_HAS\_NO\_PROC\_ENTRIES** If defined, the init code knows that no proc entries of any type all are to be created. This is used by the `pagetable` module which only exports functions;

**VMR\_MODULE\_HAS\_NO\_FILE\_ENTRIES** If defined, the init code knows that no file entries are to be created. This is only used by the `vmregress_core` module which only creates a directory;

**VMR\_HELP\_PROVIDED** If a help callback is provided, this define names the function. It takes one parameter, the procentry that is to be written to.

**NUM\_PROC\_ENTRIES** This is the number of proc entries that is to be created. Put another way, it is the number of elements in the `testinfo` array.

## 2.3.7 Proc read/write Functions

Just like the initialisation code was essentially similar over all modules, so was the proc read/write functions. The two functions `vmr_proc_read()` and `vmr_proc_write()` are now in `src/init/proc.c` and the file is `#included`. Again, the code is slightly different between modules so `#defines` determine what code is generated. The defines are as follows:

**VMR\_PROC\_READ\_CALLBACK** This is a function which is called to fill the procbuffer with information for printing to userspace. It takes just the procenry as a parameter. The callback function is responsible for opening the proc buffer and `vmr_proc_read()` will close the buffer when it is finished printing to userspace;

**VMR\_READ\_PROC\_ENDCALLBACK** This function will be called after the proc buffer is closed for modules which require additional cleanups. Currently, it is only used by the `mmap` module;

**VMR\_PROC\_WRITE\_CALLBACK** Same principle as the read callback. This callback function is expected to take two parameters, the array of arguments read from userspace and the number of parameters read;

**NUMBER\_PROC\_WRITE\_PARAMETERS** As different modules take different number of parameters when written to, this define is needed to indicate how many needs to be read;

**CHECK\_PROC\_PARAMETERS** If this callback is specified, it will be called with the arguments read from userspace for sanity checking. It should return 1 if the parameters are valid;

**PARAM\_TYPE** By default, parameters read from userspace are `ints`. This define will override that to be some other type;

## 2.4 Sense Modules

Sense modules are responsible for printing out information about the internals of the VM system. Parts of the internals are already exported via interfaces such as the `/proc/slabinfo` and much more can be inferred by the output of utilities such as **top** and **ps**. The sense modules ultimately aim to provide much more information about the VM internals.

At time of writing, there is four sense modules available. The **sizes** module prints out the size and memory usage of some internal structs. **zone** prints out information on every zone and node in the system. **kvirtual** prints out information on the kernel address space and where different areas are located. Finally, **pagemap** will print out an encoded map of all memory areas currently in use and whether the page is resident or not.

### 2.4.1 Kernel Address Space (kvirtual.o)

This module creates the proc entry `/proc/vmregress/sense_kvirtual` and concerns itself with all things vmalloc related. At time of writing, it will only print out the sizes of the zones. In later versions, it will walk through the region and print out the size of each mapping in it.

A sample output from Kernel 2.5.64 on my test box<sup>3</sup> looks like

```
root@crash:/proc/vmregress# cat sense_kvirtual
Linear Address Space
-----
o Process address space: 0x00000000 - 0xC0000000 (3072 MB)
o Kernel image reserve: 0xC0100000 + 2 * PGD      ( 16 MB)
o Physical memory map:  0xC1000000 - 0xC8000000 ( 112 MB)
o VMalloc Gap:          0xC8000000 - 0xC8800000 (   8 MB)
o VMalloc address space: 0xC8800000 - 0xFF7FE000 ( 879 MB)
o 2 Page Gap:           0xFF7FE000 - 0xFF800000 (   8 KB)
o PKMap address space:  0xFF800000 - 0xFFBFE000 (   3 MB)
o Unused pkmap space:   0xFFBFE000 - 0xFFE30000 (   2 MB)
o Fixed virtual mapping: 0xFFE30000 - 0xFFFFF000 (1852 KB)
o 2 unused pages:       0xFFFFF000 - 0xFFFFFFFF (   4 KB)
```

### 2.4.2 Pages Present/Swapped (pagemap.o)

This module creates the proc entry `/proc/vmregress/pagemap_read`. For every VMA in the process, it will print out the pagetable module from the core to print out the map. This module is used by benchmarks to read it's full address space and print out the areas been tested.

The information is encoded. While there is no script available to decode it, there is a perl helper library `VMR::Pagemap` which has a function `findmap` for finding and decoding

---

<sup>3</sup>“crash” is an uninspiring if often accurate name for my test machine

a particular VMA.

### 2.4.3 Struct Sizes (sizes.o)

This module creates the proc entry `/proc/vmregress/sense_structsizes`. It prints out the size of various structs and then attempts to print out how much memory the kernel is using on storing them. At time of writing, it is not particularly accurate at calculating memory usage and to print out information related to tasks, a kernel patch is required.

Without the patch, it is still useful for printing out struct sizes. A sample output from the authors machine with kernel patch looks like

```
root@crash:/proc/vmregress# cat sense_structsizes
Linux Kernel 2.5.64
```

```
Physical memory representation
```

```
o pg_data_t: 9600
o zone:      3072
o page:      40
o pte_chain: 32
```

```
Virtual memory representation
```

```
o vm_struct: 28
```

```
Process related structs
```

```
o task:      1568
o mm_struct: 408
o vm_area_struct: 64
```

```
Buddy related
```

```
o free_area: 12
```

```
Slab allocator related
```

```
o kmem_cache_s: 244
o slab:         24
```

```
Live System Memory Usage
```

```
nodes * 0          = 0
zones * 0          = 0
pages * 0          = 0
highpages 0 reserved 0 shared 0 swap cached 0
present_pages 0 spanned_pages 0
pte_chain          = See /proc/slabinfo for details
```

Note: Cannot show systemwide stats without `mmlist_lock` exported

The system this output is taken from does not have the kernel patch applied so the "Live System Memory Usage" cannot be filled.

## 2.4.4 Zone Statistics (zone.o)

This module prints out information via `/proc/vmregress/sense_zones` on every zone in every memory node in the system. It is particularly useful for determining how tests should be run because the number of physical pages in every zone can be determined with this module. Sample output from a live system looks like the following

```
root@crash:/proc/vmregress# cat sense_zones
Node 0
-----
ZONE_DMA                                ZONE_NORMAL
zone->present_pages =      4096  zone->present_pages =      28672
zone->spanned_pages =      4096  zone->spanned_pages =      28672
zone->free_pages     =      3317  zone->free_pages     =      18186
zone->pages_high     =         96  zone->pages_high     =         672
zone->pages_low      =         64  zone->pages_low      =         448
zone->pages_min      =         32  zone->pages_min      =         224
```

If HighMem is enabled, the zone `ZONE_HIGHMEM` will be included. The fields printed out from each zone are as follows

**size** The size of the zone in pages

**free\_pages** The number of free pages

**pages\_high** Once `kswapd` is woken up, it will remain awake until at least `pages_high` number of pages are free in the system

**pages\_low** When the number of free pages equal this watermark, `kswapd` is woken up to try and free pages

**pages\_min** At this watermark, only `GFP_ATOMIC` allocations will succeed. Other allocations will attempt to free pages for usage themselves

## 2.5 Test Modules

Test modules are responsible for testing specific code paths within the VM. They are not designed as a benchmark or to emulate real world behavior but they are useful for testing changes to the VM to make sure it still works. While this section describes how to use the modules directly, there is perl scripts in the bin directory which are a lot more friendly and collect additional information which used be used.

### 2.5.1 Test Proc Interface (testproc.o)

All modules in VM-Regress depend on the proc interface working. If it is not working, then a test can neither be started nor the results of one read. On module load, the entry `/proc/vmregress/testproc` is created and 2 pages worth of data is entered into the proc buffer. The test is quiet simple. The utility **cat** is used to read the entry and **echo** can be used to alter the size of the buffer. A sample read (clipped for verbosity) looks like

```
root@crash:/proc/vmregress# cat testproc
24 - 124: 01234567890123456789012345678901234567890 .... approx 80 more bytes
135 - 235: 0123456789012345678901234567890123456789 .... approx 80 more bytes
/* ... */
/* A lot of more data */
/* ... */
7986 - 8086: 01234567890123456789012345678901234567 .... approx 80 more bytes
8100 - 8191: .....
```

Here we can see that 2 pages of data each of 4KB have been printed. To alter the size of the buffer, echo is used

```
root@crash:/proc/vmregress# echo 5 > testproc
testproc: 5 pages allocated for proc buffer
```

Another read of the file will show 5 pages or 20480 bytes of data displayed. This module does not have a perl script as the module is only useful during development of the VM Regress proc printing core and is not of general benefit.

### 2.5.2 Test Physical Page alloc/free (alloc.o)

Physical page allocation and freeing is arguably one of the most fundamental tasks the VM must perform. They are provided by the kernel functions `__alloc_pages()` and `__free_pages_ok()`. 4 separate tests in two varieties are provided via this module. The two varieties are which `GFP_` flags are used to allocate the page. By default `GFP_ATOMIC` is used. This guarantees that the only code paths tested will be confined to `mm/page_alloc.c` as much as possible and will not sleep. If the parameter `gfp_kernel=1` is passed during module load via **insmod**, `GFP_KERNEL` will be used instead. This means that when pages



are not available, the process will enter `mm/vm_scan.c` to free pages manually and may sleep as a result.

Four proc entries for each of the tests are exposed. Each test takes two parameters. The first parameter, `nopasses` is how many times a block of pages will be allocated and then freed. The second optional parameter is how many pages to use for the test. If the second parameter is not given, the test will calculate how many pages to allocate. The number of pages allocated during each pass is printed in the report. The tests are

**test\_alloc\_fast** This will allocate pages above the `pages_high` watermark. This will test that a quick allocation through the shortest path will work.

**test\_alloc\_low** This will allocate pages until the number of free pages is between `pages_min` and `pages_low`. This should force `kswapd` to be woken up to free some pages

**test\_alloc\_min** This will allocate until the number of free pages is between 0 and `pages_min`. This will force `kswapd` to be woken up and work heavily. If `GFP_KERNEL` is used, the calling process will also work to free pages

**test\_alloc\_zero** This will stress allocations heavily. With `GFP_ATOMIC`, a number of pages half way between `free_pages()` and the zone size will be allocated. This will force heavy work by `kswapd` to try and satisfy the allocation. With `GFP_KERNEL`, it will just allocate until 0 pages are free. It will not progress more as it's very likely the VM will kill the process leaving no way to claim the pages back.

To store pointers to the pages, `vmalloc()` is used to allocate a block of kernel virtual memory to store pointers. The number of pages needed to store pointers is taken into account by the test.

This is a sample test. It passes in two parameters 5 and 10 which translates as allocating 10 pages 5 times.

```
root@crash:/proc/vmregress# echo 5 10 > test_alloc_fast ; cat test_alloc_fast
test_alloc_fast Test Results.
```

#### Test Parameters

```
o Passes:          5
o Starting Free pages: 1340
o Allocations per pass: 9
o Free page limit: 765
```

#### Test Output (Time to alloc/free)

Alloc	Free
10ms	10ms
10ms	10ms
10ms	10ms
10ms	10ms

```

10ms    10ms

Post Test Information
o Finishing Free pages: 1341
o Schedule() calls:      0
o Aborted passes       :   0

Test completed successfully

```

It shows that it took 10ms to allocate and free each time. The value for "Allocations per pass" is 9 and not 10 because 1 page is used to store pointers to each page allocated.

### 2.5.3 Test Page Faulting (*fault.c*)

Memory allocated by a user process is usually stored as pages in a Virtual Memory Area (VMA) in the processes `mm_struct`. This is true for either the heap or a privately mapped region of anonymous memory with `mmap`. This module creates a memory region with `do_mmap()` and touches all the pages within that region. For every pass of the test, it examine the pages and swaps them back in if necessary.

The tests take the same parameters as the physical page allocation tests. The names of the tests are similar

**test\_fault\_fast** Creates a region sized a number of pages that would ensure the `pages_high` watermark would not be reached

**test\_fault\_low** Same as for `alloc` except a `vma` is used to store the PTEs and pages instead of storing pointers to pages

**test\_fault\_min** Same as for `alloc` except a `vma` is used to store the PTEs and pages instead of storing pointers to pages

**test\_fault\_zero** This test is the same as the `alloc` one but is worth mentioning in more depth. The test can be used to create a region the same size as the physical memory. This will force the VM to work extremely hard to find pages.

The last test in particular is useful for comparing the current stable VM with experimental VMs. When freeing cache is not enough to free pages, the stable VM will swap out entire processes to create pages. This is particularly bad for this test as it'll force constant swapping. Recent experimental VMs try to selectively remove process pages without resorting to swapping everything. It would be interesting to see how well it handles the `test_fault_zero` test with a region close to physical memory in size.

Each of the tests output four columns of interest, Pass, Refd, Present and Time. Pass is which pass through memory this is. Refd is how many pages were swapped in this pass. Present is how many physical pages were still in memory after the pass and Time is how long it took the pass to complete.

Sample test output from a live machine follows

```
root@crash:/proc/vmregress# echo 2 > test_fault_fast ; cat test_fault_fast
test_fault_fast Test Results.
```

#### Mapped Area Information

```
o address: 0x4019B000
o length: 1892352 (462 pages)
```

#### Test Parameters

```
o Passes: 2
o Starting Free pages: 1227
o Free page limit: 765
o References: 462
```

#### Test Results

(Pass	Refd	Present	Time)
0	462	462	10ms
1	0	462	0ms
2	0	462	0ms

#### Post Test Information

```
o Finishing Free pages: 1227
o Schedule() calls: 0
o Failed mappings: 0
```

Test completed successfully

Here a region of memory 462 pages large was created. They were all swapped in during the first pass (Refd = 462) and none of them were swapped out. It took 10ms. Because memory is under no pressure, the rest of the passes had no work to do.

After the test results, an encoded map of the memory region will be printed out. Every character in the map is 4 pages. The lower four bits represent the pages. If a page is present, the bit is set. Two of the higher bits are set so that the map is semi-readable to humans. The map can be used to produce a graph with **gnuplot**.

For this test on old kernels, the graph is not very interesting. All the pages are either all present, all swapped or in the process of been all swapped. This is because of how the page replacement algorithm works. In later kernels, especially with rmap and different reference patterns, it is expected this type of test will show the performance of the replacement algorithm.

This is a sample map of what 32 pages looks like.

```
BEGIN PAGE MAP 0x40156000 - 0x40176000
????????
END PAGE MAP - 32 pages of 32 present
```

The first line shows the region that was mapped. The question marks represent the

bit pattern 0x00111111 to show all pages are present.

## 2.6 Benchmark Modules

This section covers any modules available for benchmarking purposes. At time of writing, there is only one called **mmap** which is focused toward benchmarking mmap with read/write tests on anonymous or file mapped regions.

Benchmark modules are quite different in what they intend to provide than the test modules. Test modules run a particular test to see if it succeeded or not. They maintain internal state and cannot be interrupted. Bench modules run a particular test to see how well it performed. It does not maintain internal state and depends on a userland program to collect and userlevel information while the kernel module will provide any relevant kernel data.

As bench modules are a bit involved, it is highly recommended the perl scripts are used to drive them as a lot more information may be collected and reported on.

### 2.6.1 Bench mmap Performance (mmap.o)

This is intended for scripts that want to benchmark page faulting and replacement code. It focuses on the use mmaped regions, either anonymous or file descriptors. 6 proc entries are provided.

**mapanon\_open** takes takes one parameter, the number of bytes to map. It maps a region of anonymous memory of the requested size private to the process. The address is stored in **mapanon\_addr** for the process until it is picked up. Because of this, it is very important **map\_addr** is read after writing to **mapanon\_open**;

**mapfd\_open** takes 5 parameters. The length to map, the protection flags, the mapping flags, the file descriptor and the offset within the file to map. This is directly related to the parameters mmap takes;

**map\_read** takes two parameters. The address to reference and the length to read. The bytes are the location will be read but not returned. The module is only interested in affecting the pages;

**map\_write** takes two parameters as well except it will write instead of read the address. Remember that if the file is memory mapped and mapped **MAP\_SHARED**, it will be overwritten;

**map\_close** takes two parameters, the address to unmap and the length. This is the equivalent of **munmap()**;

**map\_addr** is a hack and a weird one at that. When a caller uses open to create a mapped region, there is no way to return the address. Returning the addr through procs gets lost in the ether. What happens is that when an address is opened, it is placed in the **map\_addr** and the buffer locked for the calling PID until the proc entry can be read.

This interface was used instead of using the standard mmap interface for two reasons. First, kernel information can be collected from the module later if things like page fault tracing is available. It was considered to have the infrastructure in place to allow full kernel data collection sooner rather than later.

Second, not all languages use mmap in the same way. Perl does not natively provide mmap so the only way to be sure how mmap and the kernel is to get a kernel module to do the work.

# Chapter 3

## Userland Scripts

The kernel modules are intended to give fine grained control and information about the kernel but because of their development complexity, the tests are run from userspace. The scripts are able to decide which kernel modules they require and load them for the duration of the procedure, collect and necessary information and print a report.

The `sense` and `test` may be used in isolation. They will print out their results to the `proc` entries without any trouble but there is still a lot more information that may be gathered from userspace. Driver scripts are provided to load the relevant kernel modules, pass the appropriate parameters, collect additional information and print a report.

The benchmark are different as in themselves, they are not very useful like the `sense` and `test` modules. Benchmarking is a lot more involved and both userland scripts and kernel modules are required to collect any meaningful data.

Each of the scripts have comprehensive man pages but many share the same parameters. The four most common ones are:

- help** Print a small help message;
- man** Print a man page. Requires the **perldoc** package;
- passes** Some tests can be run multiple times with this parameter
- nounload** Scripts know what kernel modules need to be loaded to perform the test or operation. Normally, modules that are loaded are unloaded again at the end of the script which is sometimes undesirable. This switch prevents modules been unloaded
- output** This means one of two things. If the specified parameter is a directory, then suitable filenames will be chosen for the report there. If it is a filename, it will be used as a prefix for any generated files

Regrettably, the presence of the script itself will affect the outcome of the test but it cannot be avoided and the scripts try and reduce the impact as much as possible. All the scripts come with comprehensive man pages so will not be discussed in as much detail here.

## 3.1 Physical Page Test

The **test\_alloc.pl** script is responsible for running the physical page alloc test with the alloc module. Additionally, it will collect the output from vmstat.

Usage:

```
test_alloc.pl [options]
```

Options:

<code>--help</code>	Print help messages
<code>--man</code>	Print man page
<code>--testfast</code>	Run test with pages allocated above <code>pages_high</code>
<code>--testlow</code>	Run test between <code>pages_low</code> and <code>pages_min</code>
<code>--testmin</code>	Run test between 0 and <code>pages_min</code>
<code>--testzero</code>	Run test with more pages than physical memory
<code>--nounload</code>	Do not unload kernel modules
<code>--passes</code>	Number of passes to make on test
<code>--size</code>	Optionally specify the number of pages to use
<code>--output</code>	Output files prefix

The report consists of the output from the proc interface, a graph of vmstat output and information about the host machine. The four test switches correspond directly to the four proc entries that the alloc module creates. The number of pages used for the test may be overridden with `--size`.

## 3.2 Page Fault Test

The script **test\_fault.pl** is a front-end to the **fault** module and follows a similar principle to the test described in the previous section

Usage:

```
test_fault.pl [options]
```

Options:

<code>--help</code>	Print help messages
<code>--man</code>	Print man page
<code>--testfast</code>	Run test with pages allocated above <code>pages_high</code>
<code>--testlow</code>	Run test between <code>pages_low</code> and <code>pages_min</code>
<code>--testmin</code>	Run test between 0 and <code>pages_min</code>
<code>--testzero</code>	Run test with more pages than physical memory
<code>--passes</code>	Number of passes to make on test
<code>--size</code>	Optionally specify the number of pages to use
<code>--output</code>	Output files prefix

The report is essentially the same as the alloc test.



### 3.3 Anonymous and File Based Faulting Benchmark

Benchmarking the mmap performance is broken up into two parts. Generating the page reference data and the actual test. It is recommended the reference data is generated in advance so the same test can be run repeatedly and because it is very time consuming to generate a large test dataset. The generated data set is stored on disk.

#### 3.3.1 Generating Page Reference Data

Generating realistic page reference data is an involved task, far more involved than it would appear at first glance. VM Regress does not make much attempt at generating realistic data as of time of writing and is more concerned with providing hooks for users to test their own data. This will change over time as different sets of workloads are tested.

The script `generate_references.pl` is responsible for generating references.

Usage:

```
generate_references.pl [options]
```

Options:

<code>--help</code>	Print help messages
<code>--man</code>	Print man page
<code>--pattern</code>	Page reference pattern (default: linear)
<code>--size</code>	Size of area to benchmark with
<code>--references</code>	Number of references to generate
<code>--output</code>	Output filename

The description of each as taken from the man page are as follows

- `--pattern` This is the pattern to reference pages at. There is three types, **linear**, **smooth\_sin** and **random**. **linear** will refer to pages from beginning to end. **smooth\_sin** when the page reference count graph is generated will show a sin wave pattern. It should be noted this is nothing resembling real page references. The last is random which generates a pile of random references of no particular order.
- `--size` This is the size in pages the memory area been referenced is.
- `--references` This is the number of references within the range to make. Note that to get a noticeable sin curve with **smooth\_sin**, the number of references will need to exceed the range considerably

The size of the area and the number of references is outputted to the first line of the output file for convenience. The rest of the file is one page reference per line.

### 3.3.2 Randomizing Page Reference Data

The **generate\_references.pl** script produces data that is very linear in nature. For example, the **smooth\_sin** generates references that refers to some pages more than others but it performs it in page order. This equates to a linear scan through memory. The script **randomize\_references.pl** will randomize the output to **generate\_references.pl**.

Usage:

```
generate_references.pl [options]
```

Options:

<b>--help</b>	Print help messages
<b>--man</b>	Print man page
<b>--input</b>	Input reference file
<b>--output</b>	Output reference file

This is convenient when checking how pages are replaced with respect to frequency as opposed to age.

### 3.3.3 Generating A File To mmap()

If the benchmark is to be used to benchmark with a file mapped rather than anonymous memory, a file has to be created of the correct size. Note that the scripts and benchmark makes no attempt to verify the file is correct. It is left to the user to use a proper file.

The easiest way to create the file is by using **dd** with **/dev/zero** to generate a zero filled file. To create a file 20000 pages large on an x86, the command is

```
dd if=/dev/zero of=/home/user/filemap bs=4096 count=20000
```

It is very important that a file is created that is large enough to be referenced. If a file is created that is 1MiB in size and page reference data is generated for an area 2MiB in size, the kernel will oops and kill the benchmarking proces with a **SIGBUS**. This happens because **copy\_from\_user()** will be using an invalid region.

### 3.3.4 Running The Benchmark

The **bench\_mmap.pl** is responsible for running the benchmark. It gathers a number of different types of information and produces a report when it is complete. It requires the **Time::HiRes** Perl module, a recent version of **gnuplot** and the **imagemagick** tools to be available.

The first section of the report details the test parameters including the kernel version, the reference data used, whether it was anonymous memory or file mapped and whether is was a read or write test.

The second section details information about the mapped region including it's size, how many pages were referenced, fastest/slowest access and so on.

The third section gives the zone statistics before and after the test to give the reader some indication what sort of pressure was placed on the system.

The fourth section is a number of graphs which give different indications about the performance during the test. The graphs are as follows

**Page Access Times** This shows how long it took to access each page over time. The y axis shows the length of time to access the page. The x axis is which number reference it was.

**Page Index Reference over Time** Each each reference made, this graph shows what page was been accessed. The y axis is the size of the memory region been referenced. The x axis is the number reference. This graph can illustrate what sort of reference data was been used. That is, was the references smooth, burst or random in nature.

**Page Age/Presence Map** This graphs how old each page was and if it was in memory or not. The x axis is the page within the region mapped. The green line shows the age of the page. The red line will be half the maximum y value if the page was present and 0 if it was swapped out. This indicates if the page swap algorithm was able to keep the correct page in memory based on age.

**Page Reference/Presence Map** This is similar except it graphs based on reference count rather than age.

**VMStat output** During the test, vmstat is running. This graph shows graphs four values from the output. **swpd** is the amount of virtual memory used. **free** is the number of free pages in the system. **buff** is the amount of memory used in buffer caches. **cache** shows the amount used in the page cache.

The last three sections of the report are the output from `/proc/cpuinfo`, `/proc/meminfo` and the raw output of `vmstat`. This is for the reader so they know what sort of system the rest was run on.

Usage:

```
bench_mapanon.pl [options]
```

Options:

<code>--help</code>	Print help messages
<code>--man</code>	Print man page
<code>--size</code>	Size of area to benchmark with
<code>--refdata</code>	Alternatively, use this source file of reference data
<code>--write</code>	Boolean to indicate if a write test should be performed
<code>--pattern</code>	Page reference pattern (default: linear)
<code>--filemap</code>	File to memory map (default: use anonymous memory)
<code>--references</code>	Amount of references to generate
<code>--passes</code>	Number of times to read reference data
<code>--time_maxy</code>	The maximum yrange for the time page reference graph
<code>--nounload</code>	Do not unload kernel modules
<code>--output</code>	Output filename (extensions appended)

Four sets of data are preserved for future analysis. The first `prefix-time.data` is how long each page reference took in microseconds. The first column is the number reference, the second column the time. The second data file is `prefix-age.data`. For each page in the mapped region, the second column will be its age. The third data file is `prefix-pagemap.data` shows what pages were present in memory. If it is 0, it is absense and otherwise it is present. The last file is `prefix-refcount.data` which contains how many times each page was referenced. If a set of page references was not provided, the generated file will be stored in `prefix-pagereferences.data` but it is recommended data is prepared with the `generate_references.pl` script.

For each of these files, the prefix is determined by the `-output` switch. By default it will be `./mapanon`.

# Chapter 4

## Perl Support Libraries

The benchmark scripts rely heavily upon Perl libraries to do most of the work. This is aimed at allowing a developer to easily integrate new tests with the existing framework. This section will cover the eight libraries available. They are all located under `bin/lib/VMR`.

### 4.1 VMR::External

This module is concern with handling data from external programs such as **vmstat**, **oprofp** and any other program the user wishes to capture data from. It presumes that only one instance of each program is opened. If that presumption changes, the caller will have to start tracking the file handles returned from `openpipe` themselves as `@HANDLES` will be useless

Three functions are provided, `openexternal()`, `readexternal()` and `closeexternal()`.

```
##
# openextrnal - Opens a pipe to an external program
# @program: The name of the program to exec
# @arguments: Arguements to pass to the program
#
# This function will exec an external program and record a handle to it that
# may be read later. Because of the weird way PERL handles file handle names,
# the function is limited to execing 3 programs. If more are required, add
# new pipe names below

##
# readexternal - Read from an external pipe
# @program - The program name to read
#
# This will read a pipe to a program that was execed using openexternal
```

```
##
# closeexternal - Close a pipe to an external program
# @program - Name of the program to close
```

## 4.2 VMR::File

This is concerned with File IO related tasks. At time of writing, this is confined to creating temporary file names and reading/writing proc entries.

```
##
# mktempname - Make a temporary filename
# @name: Name of the program creating the tempname (optional)
#
# This function will return the name of a file that is unique. It is not bullet
# proof and doesn't guarantee two callers will create the same temp name at the
# same time but is sufficient for current purposes
```

```
##
# readproc - Read a proc entry
# @procentry: Name of the proc entry to read
#
# The function will return the entire contents of a proc entry. If a full
# path is not provided, the entry is presumed to be in /proc/vmregress
```

```
##
# writeproc - Write to a proc entry
# @procentry; Name of the proc entry to write
# @write: Information to write to it
#
# The contents of $write will be written to the specified procentry. If a
# full path is not provided, the entry is presumed to be in /proc/vmregress
```

## 4.3 VMR::Graph

This is a frontend to **gnuplot**. It will take a given data source and treat it different depending on the type of data it is. A large number of parameters have to be passed.

```
##
#  gnuplot - Call gnuplot to generate a graph
#  @type: Type of source data
#  @title: Title for the graph
#  @xrange: The X range described as from:to
#  @yrange: The Y range described as from:to
#  @output: Output PNG file
#  @ds1: The first data source
#  @ds1name: Name of the data been graphed
#  @ds2: The second data source
#  @ds2name: Name of the second data been graphed
#
#  This opens a pipe to gnuplot and prepares to plot. It understands a number
#  of different data types. They are vmstat, PageReference and default.
#  vmstat takes the output of vmstat as input. PageReference takes two
#  data sources, the page reference count and page presence. The last
#  default will just plot a normal graph
```

The library has two internal functions for parsing vmstat data and then cleaning up the temp files generated.

## 4.4 VMR::Kernel

This module is concerned with the VM Regress kernel modules. It checks for the existence of certain kernel modules. If they are not available, it attempts to load them. If it fails, the caller is killed

```
##
#  checkmodule - Check if a particular module is loaded. If not, load it
#  @name: Name of the module to check
```

## 4.5 VMR::Pagemap

The pagemap and pagetable modules provided encoded information which shows what pages are present within an address space. This module provides two functions for decoding this information. findmap() will take the full proc output and find the desired address space and calls decodemap() to decode it.

```
##
# findmap - Find a map belonging to a particular address and decode it
# @proc: The full output from the proc entry
# @addr: The address of interest
# @mark: Used by decodemap
#
# If no addr is provided, the first map occurred is decoded and returned
# to the caller. It returns in order
#
# $range: The address range of the map decoded
# $decode: A line separated file showing pages and if it is present
# $present: The number of present pages
# $total: The total number of pages

##
# decodemap - Decode the map provided by the pagemap module
# @map: String provided by pagemap
# @mark: What to print out for page presense
#
# This will take the encoded string from the proc entry and print out a set
# of lines, each containing the page offset and if the page is present or not.
# If present, $mark is printed, otherwise 0.
```

## 4.6 VMR::Reference

This module is responsible for producing page reference information. The caller names the reference pattern they are looking for and they are returned an array. The module doesn't determine whether the accesses are read or write. That is for the caller to decide.

If a developer has proper reference data or can generate data of a pattern known to resemble program behaviour, this is the module that should be updated. The module already contains two internal functions for generating linear references and one that generates a sin curve when the page reference count is plotted



```
##
# generate_references - Select which worker function to provide page references
# @pattern: String denoting which reference pattern to use
# @references: Number of references to generate
# @range: Number of pages that can be addressed
# @burst: Boolean to denote if the pattern should block read pages
#
# Two patterns can be generated. linear will reference each page in the address
# space in order until the number of required references is generated.
# smooth_sin will generate a set of page references that looks like a sin
# curve when plotted.
```

## 4.7 VMR::Time

This module is dedicated to collecting timing information. It is used when microsecond accuracy is required. It exports two functions, `gettime()` and `difftime()`. `gettime()` returns a token representing the current time. The caller should not be concerned with it and just pass it blindly to `difftime`. `difftime` returns two values, the elapsed time in microseconds and the new time.

```
##
# gettime - Return a token representing the current time
#
# The token should be passed to difftime directly to get the elapsed time
# in microseconds

##
# difftime - Return the elapsed time since the token was taken
#
# returns the difference and a token representing the current time
```

## 4.8 VMR::Report

This is a very simple module dedicated to report generating. The code is very basic and is essentially a glorified collection of print statements so that HTML is not scattered all over other scripts and the format of reports remains relatively constant.

```
##
# reportHeader - Print the HTML header and title
# @title: Title of report
```

```
##
# reportPrint - Print a string verbatim to the report
# @string:      String to print

##
# reportZone - Print out the current node/zone information
# @append: String to append to title (e.g. Before Test)

##
# reportTest - Print out the test results
# @result:Results

##
# reportGraph - Show a graph
# @caption:      Caption to give the graph
# @path:         Path to output image
# @psfile:       Postscript source of the image
# @pngfile:      PNG file of the image

##
# reportEnvironment - Print out information on the test environment
# @vmstat: vmstat output

##
# reportFooter - Print the footer of the report page

##
# reportOpen - Open a new report
# @filename: Filename of report to open

##
# reportClose - Close the report
```

# Chapter 5

## Interpreting Results

This section will eventually cover how to interpret and evaluate results so they are not a meaningless collection of numbers and graphs. The information is incredibly sparse because the tools required to perform decent statistical analysis are simply not available yet. At the moment though, all available information is presented in the reports.